AI Assisted Coding
Lab Assignment 10.5

Name: K. Charan Yadav

Hall Ticket no: 2303A52367

Batch No: 20

Task -1:
Prompt:
Refactor the given Python program to improve code readability by using proper variable naming conventions and implement exception handling to safely manage runtime errors such as division by zero. The program should perform addition and division operations and display appropriate outputs.

```
# Task - 1: Variable Naming Issues
def add_numbers(first_number, second_number):
    return first_number + second_number
print(add_numbers(10, 20))

# Task - 2:  Missing Error Handling
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero. Please provide a non-zero divisor.")
        return None
print(divide(10, 0))
```

OUTPUT:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
ignment-10.5.py"
30
```

Justification:
Applying meaningful variable names enhances code clarity and maintainability, while error handling improves program robustness by preventing crashes and ensuring safe execution when invalid inputs occur.

Task 2:
Prompt: Enhance the Python division function by adding exception handling using a try-except block to manage division by zero errors. The program should display a meaningful error message and prevent the program from crashing when an invalid divisor is provided.

```python
# Task - 2:  Missing Error Handling
def divide(a, b):
    try:
        return a / b
    except ZeroDivisionError:
        print("Error: Cannot divide by zero. Please provide a non-zero divisor.")
        return None
print(divide(10, 0))
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding> & C:/Users/
ignment-10.5.py"
Error: Cannot divide by zero. Please provide a non-zero divisor.
None
```

Justification: Exception handling improves program stability and user experience by safely managing runtime errors. Handling ZeroDivisionError ensures the program executes smoothly even when incorrect input values are given.

Task-3:

Prompt: Develop a Python program to calculate the total marks, average marks, and grade of a student using a list of marks. The program should validate the input to ensure it is a non-empty list containing only numeric values and handle invalid inputs by displaying appropriate error messages.

```python
# Task - 3: Student Marks Processing System
def calculate_student_grade(marks):
    """Calculate the total, average, and grade for a student.
    Args:
        marks (list): A list of student marks
    Returns:
        tuple: (total, average, grade) or (None, None, None) if invalid input"""
    if not isinstance(marks, list) or len(marks) == 0:
        print("Error: Marks must be a non-empty list.")
        return None, None, None
    if not all(isinstance(mark, (int, float)) for mark in marks):
        print("Error: All marks must be numeric values.")
        return None, None, None
    total_marks = sum(marks)
    average_marks = total_marks / len(marks)
    grade = determine_grade(average_marks)
    return total_marks, average_marks, grade
def determine_grade(average):
    """Determine letter grade based on average marks."""
    if average >= 90:
        return "A"
    elif average >= 75:
        return "B"
    elif average >= 60:
        return "C"
    else:
        return "F"
# Test the function
student_marks = [78, 85, 90, 66, 88]
total, average, grade = calculate_student_grade(student_marks)
if grade:
    print(f"Total: {total}, Average: {average:.2f}, Grade: {grade}")
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
ignment-10.5.py"
Total: 407, Average: 81.40, Grade: B
```

Justification:

Input validation improves program reliability by preventing incorrect data from being processed. Proper error handling ensures accurate grade calculation and enhances robustness by avoiding runtime errors.

Task-4:

Prompt:

Use AI assistance to enhance the given Python factorial function by adding clear docstrings and meaningful inline comments. The documentation should explain the function's purpose, parameters, return value, possible exceptions, and working logic to improve code understanding.

```python
# Task - 4: Use AI to add docstrings and inline comments to the following function.
def factorial(n):
    """Calculate the factorial of a given number.
    This function computes the factorial of a non-negative integer n,
    which is the product of all positive integers less than or equal to n.
    For example, factorial(5) = 5 * 4 * 3 * 2 * 1 = 120.
        n (int): A non-negative integer for which to calculate the factorial.
        int: The factorial of n.
    Raises:
        ValueError: If n is negative.
    Example:
        >>> factorial(5)
        120
        >>> factorial(0)
        1"""
    # Initialize result to 1 (multiplicative identity)
    result = 1
    # Iterate from 1 to n (inclusive) and multiply each value to result
    for i in range(1, n + 1):
        result *= i
    # Return the computed factorial
    return result
print(f"Factorial of 5: {factorial(5)}")
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding> &
ignment-10.5.py"
Factorial of 5: 120
```

Justification:

Docstrings and inline comments improve code readability and maintainability by clearly explaining functionality and logic. Proper documentation helps developers understand, reuse, and debug the code efficiently.

Task-5:

Prompt:

Develop a Python program to validate a user password based on security requirements such as minimum length, presence of uppercase and lowercase letters, digits, and special characters. The program should return appropriate validation messages indicating whether the password is strong or specifying the missing requirement.

```
# Task - 5: Password Validation System (Enhanced)
def validate_password(password):
    """Validate password against security requirements.

    Args:
        password (str): The password to validate

    Returns:
        tuple: (is_valid, message) where is_valid is bool and message is str"""
    # Check minimum length requirement
    if len(password) < 8:
        return False, "Password must be at least 8 characters long."
    # Check for at least one uppercase letter
    if not re.search(r'[A-Z]', password):
        return False, "Password must contain at least one uppercase letter."
    # Check for at least one lowercase letter
    if not re.search(r'[a-z]', password):
        return False, "Password must contain at least one lowercase letter."
    # Check for at least one digit
    if not re.search(r'\d', password):
        return False, "Password must contain at least one digit."
    # Check for at least one special character
    if not re.search(r'[!@#$%^&*()_+\-=\[\]{};:\'",.<>?/\\|`~]', password):
        return False, "Password must contain at least one special character."
    return True, "Strong password!"
# Get user input and validate
user_password = input("Enter password: ")
is_valid, validation_message = validate_password(user_password)
print(validation_message)
```

Output:

```
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding> &
ignment-10.5.py"
Enter password: Charan@9948
Strong password!
PS C:\Users\chara\OneDrive\Desktop\Ai-Assisted Coding>
```

Justification:

Password validation enhances application security by enforcing strong password policies. Implementing validation checks helps prevent weak passwords and protects systems from unauthorized access.